Compactage

L'utilisation de listes chaînées est assez gourmande en mémoire, et nécessite des allocations dynamiques coûteuses en temps

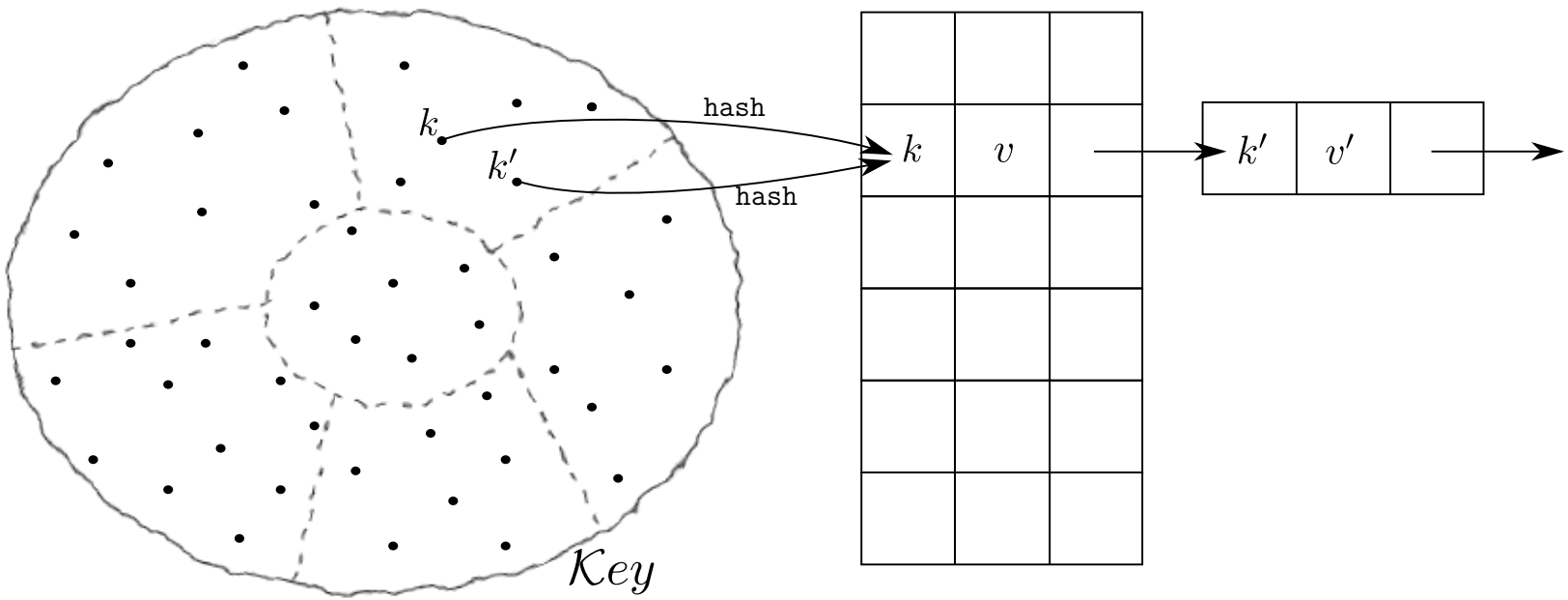
Si on a n éléments dans une table de hachage de taille n, on aimerait ne pas avoir à utiliser plus de n fois la taille d'une clef et de la valeur correspondante.

- On utilise le tableau lui même pour stocker les associations qui seraient à la suite dans la liste chaînée

On parle alors de **table de hachage fermée**

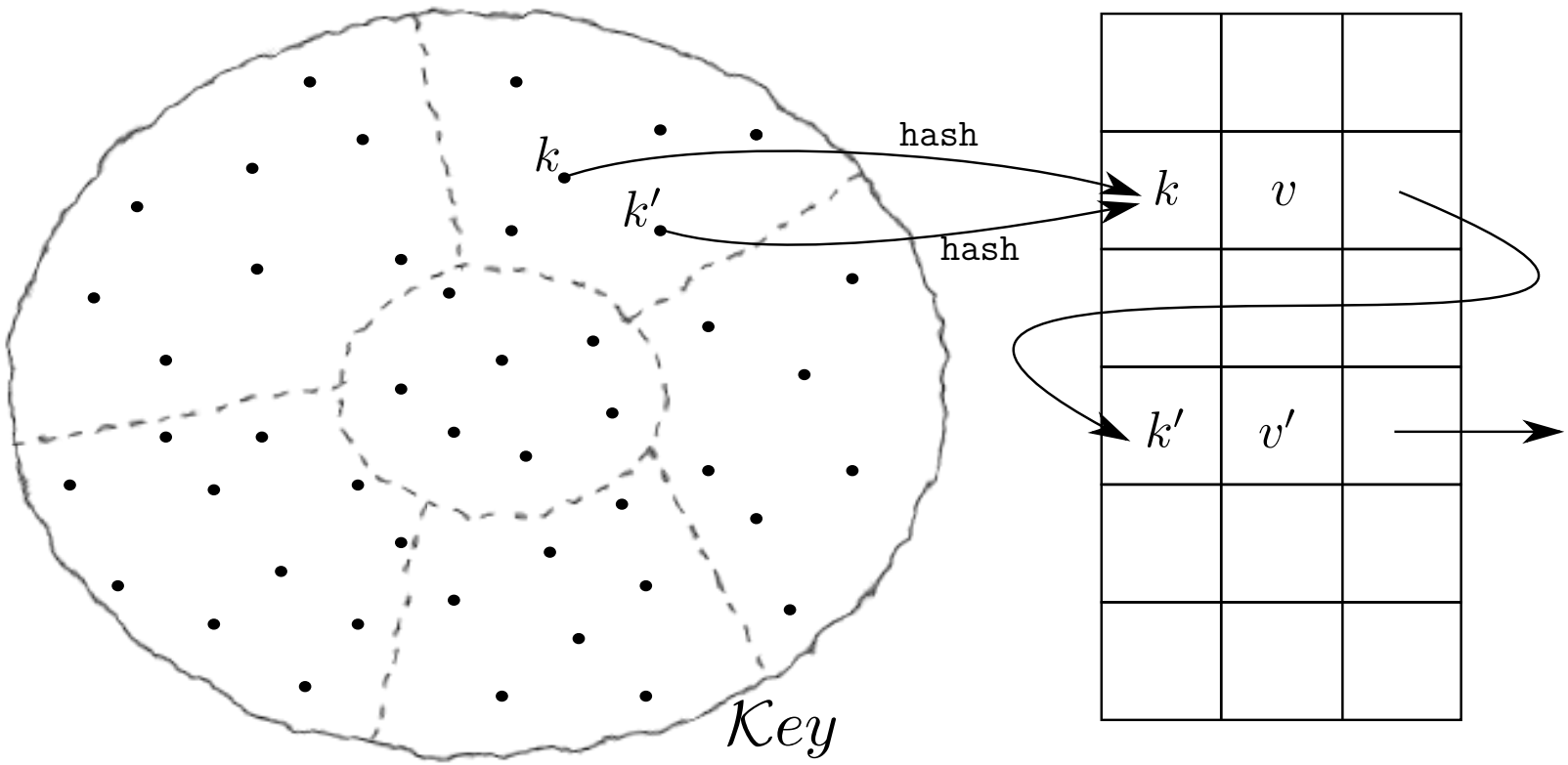
1re idée

Le tableau ne contient pas des pointeurs vers des listes chaînées, mais les premiers éléments de la liste



2e idée

On utilise le tableau pour les éléments suivants de la liste chaînée

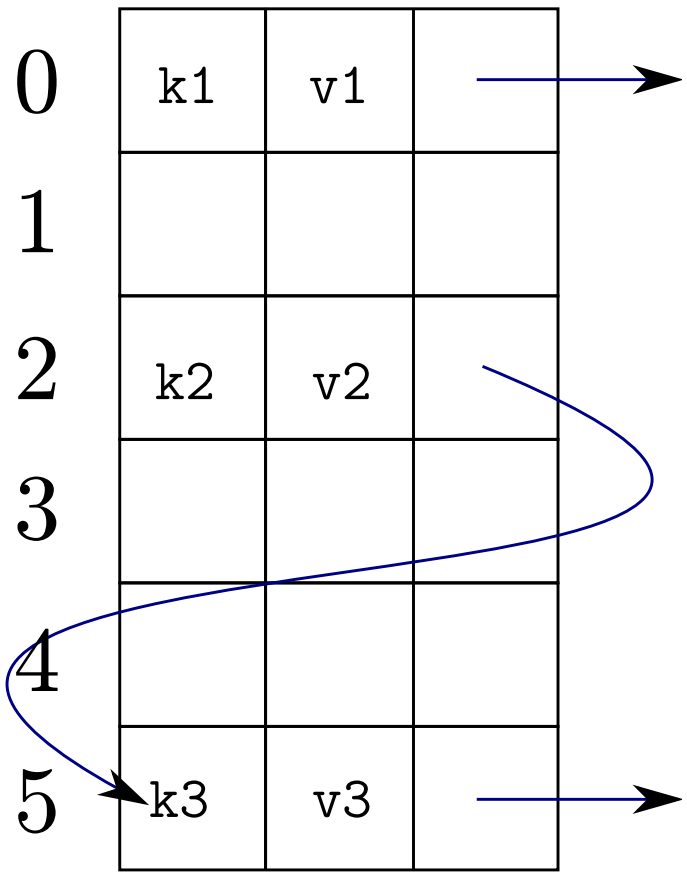


Rechercher : exemple 1

`rechercher(d,k)`

- `hash(k) = 1`

Pas trouvée

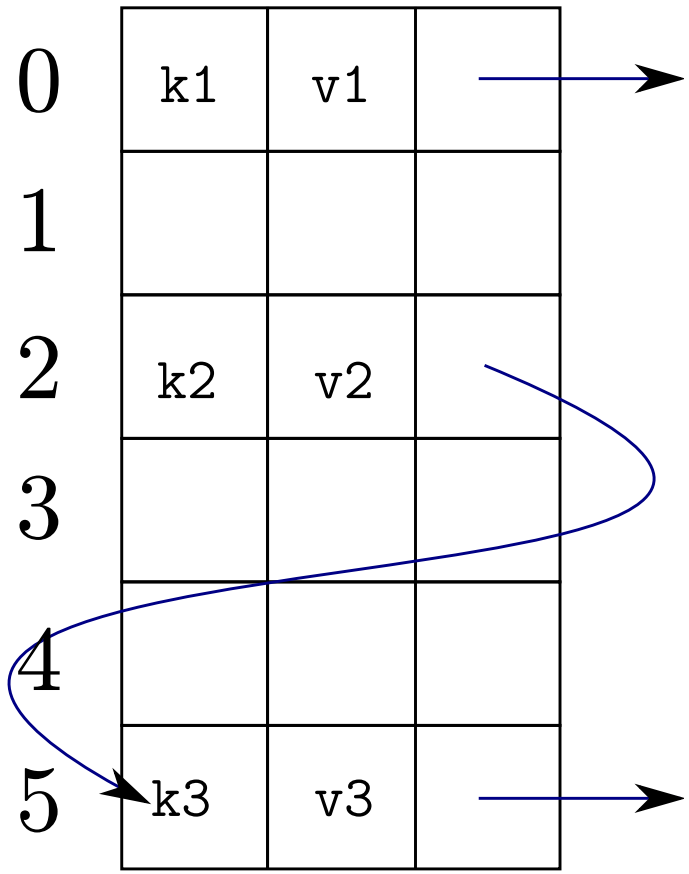


Rechercher : exemple 2

`rechercher(d,k3)`

- `hash(k3) = 2`

On suit les liens jusqu'à trouver la clef (ou non)

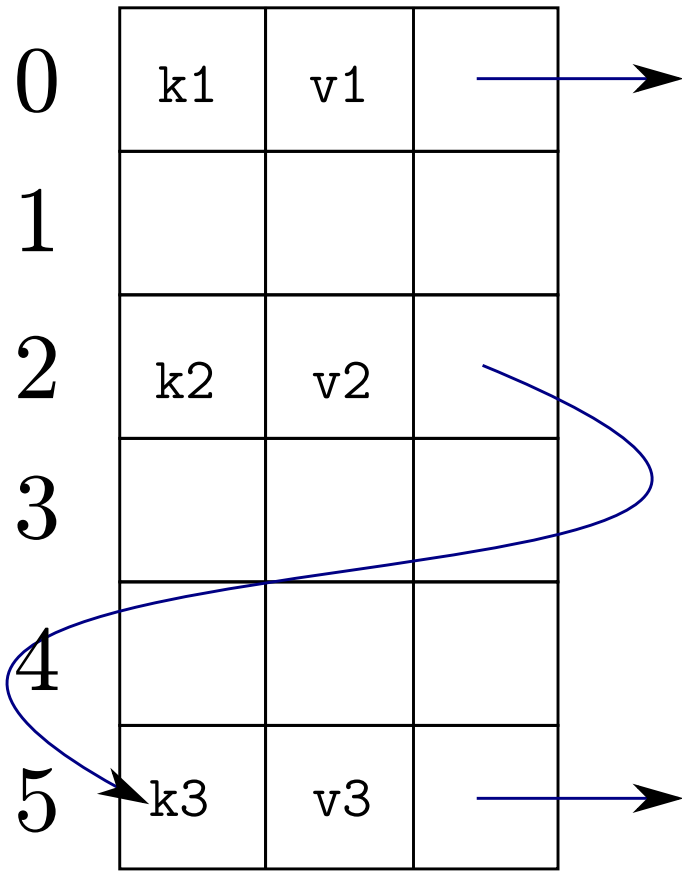


Rechercher : exemple 3

`rechercher(d,k')`

- `hash(k')` $= 5 \neq$ `hash(k3)`

Pas trouvée



Orbite

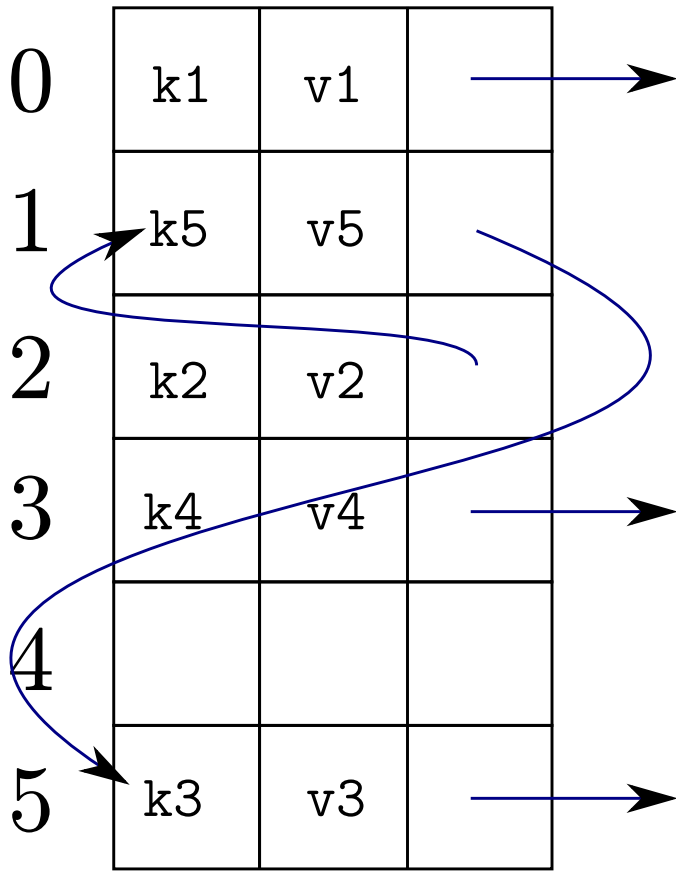
Deux associations `(k,v)` et `(k',v')` sont dans la même **orbite** si `hash(k) = hash(k')`

Orbites :

`{(k1,v1)}`

`{(k2,v2);(k5,v5);(k3,v3)}`

`{(k4,v4)}`



Recherche

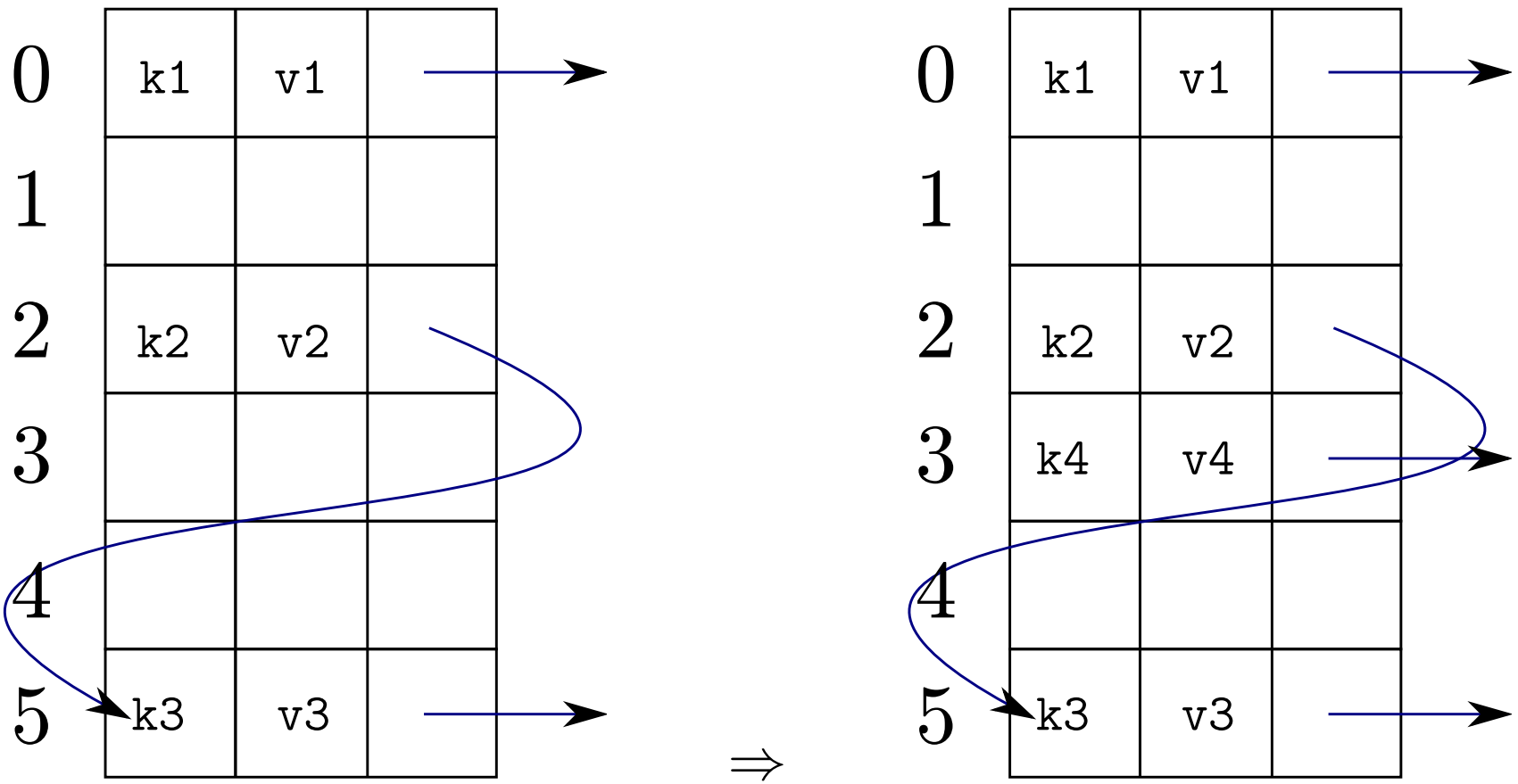
`rechercher(d,k)`

- on calcule `hash(k)`
- si la case en `hash(k)` est vide
  - clef non trouvée
- si la case en `hash(k)` contient une clef de hachage différent de `hash(k)`
  - clef non trouvée
- sinon
  - si la clef est `k`, on renvoie la valeur
  - sinon on va à la case indiquée dans le successeur

Insertion : exemple 1

`inserer(d,k4,v4)`

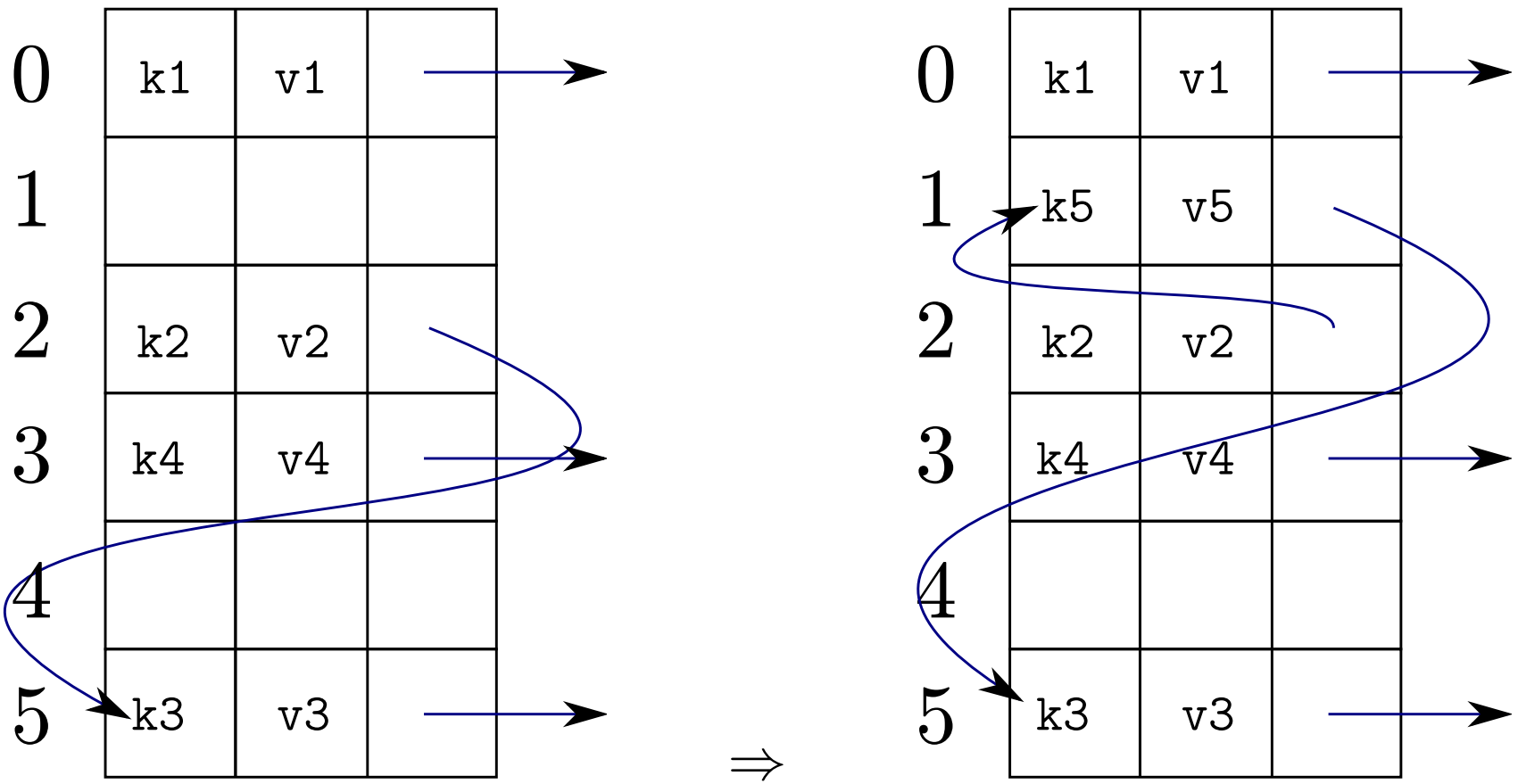
- `hash(k4) = 3`



Insertion : exemple 2

`inserer(d,k5,v5)`

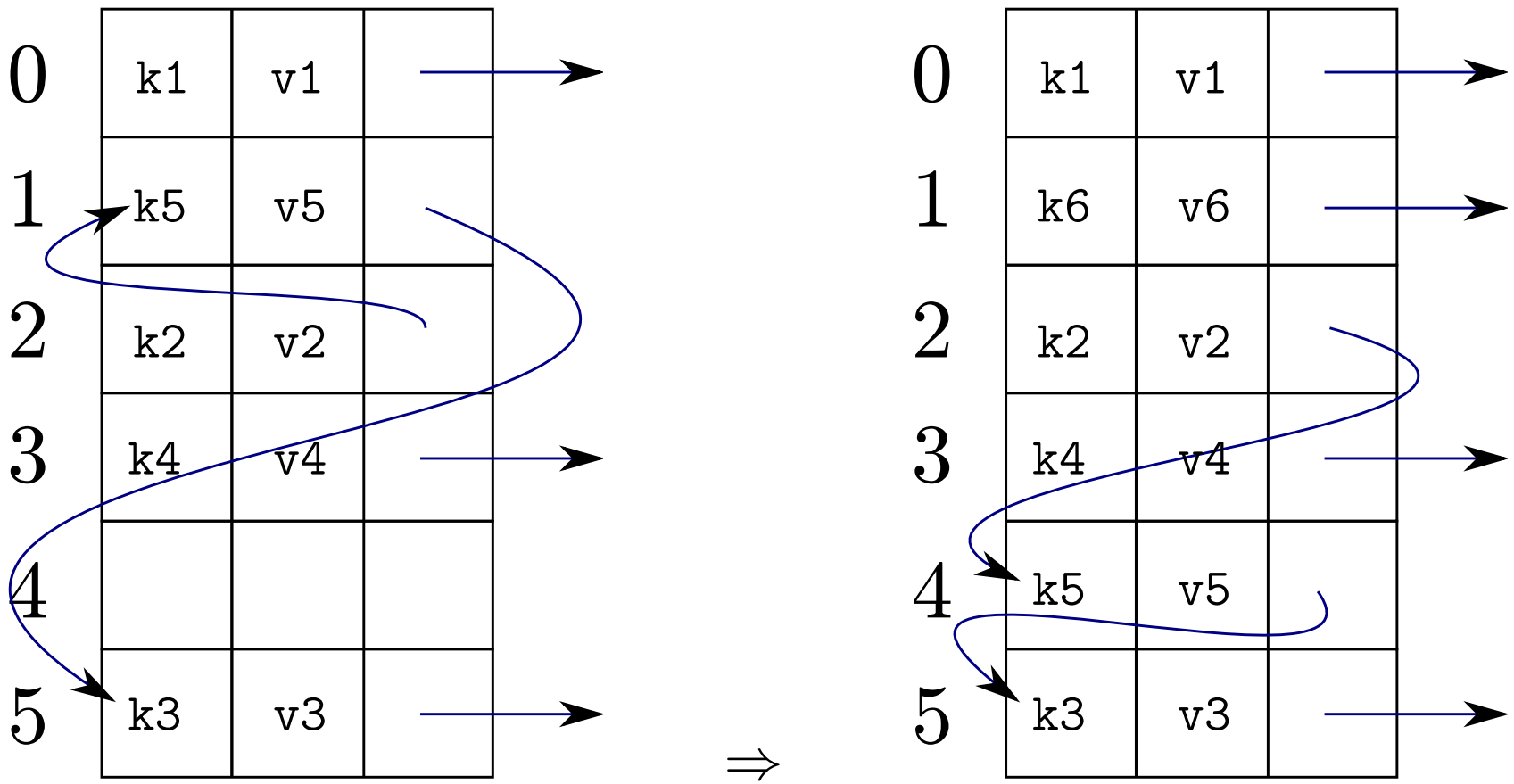
- `hash(k5) = 2`



Insertion : exemple 3

`inserer(d,k6,v6)`

- `hash(k6) = 1`



Insertion

`inserer(d,k,v)`

- on calcule `hash(k)`
- si la case `hash(k)` est libre, on y met `k,v`
- si la case `hash(k)` est occupée par une clef `k'` avec `hash(k')=hash(k)`
  - on trouve une case vide à la position j
  - on rempli la case j avec `k,v`
  - on met comme successeur en j le successeur en `hash(k)`
  - on met j comme successeur en `h(k)`

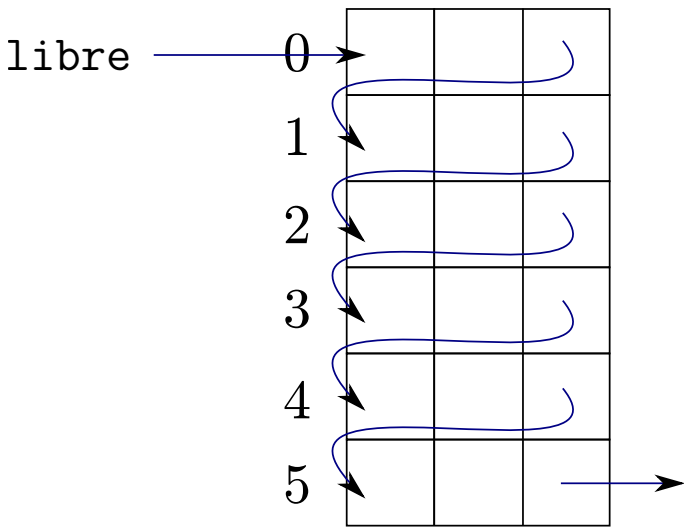
Insertion (suite)

- sinon on a `k',v',s'` en `h(k)` avec `hash(k')` $\neq$ `hash(k)`
  - on met `k,v` en `hash(k)`
  - on suit les successeurs à partir de `hash(k')` jusqu'au prédécesseur j de `hash(k)`
  - on met `k',v',s'` dans une case libre
  - on met à jour le successeur de j sur cette case

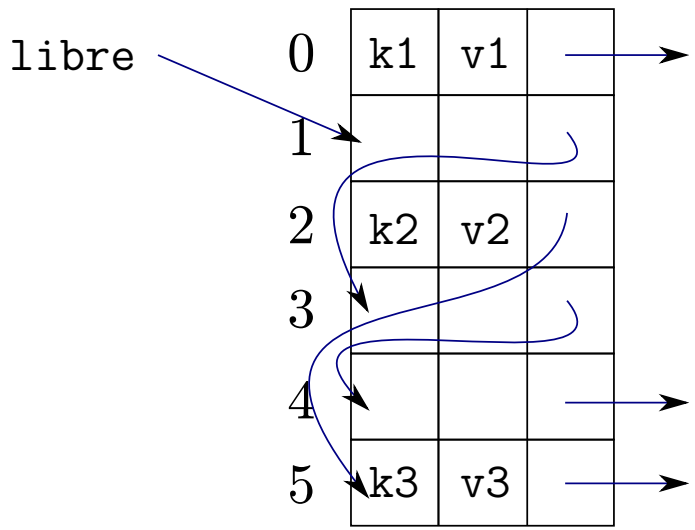
Gestion des cases libres

On a un pointeur vers la première case libre, et on utilise le successeur pour avoir les suivantes

Initialement



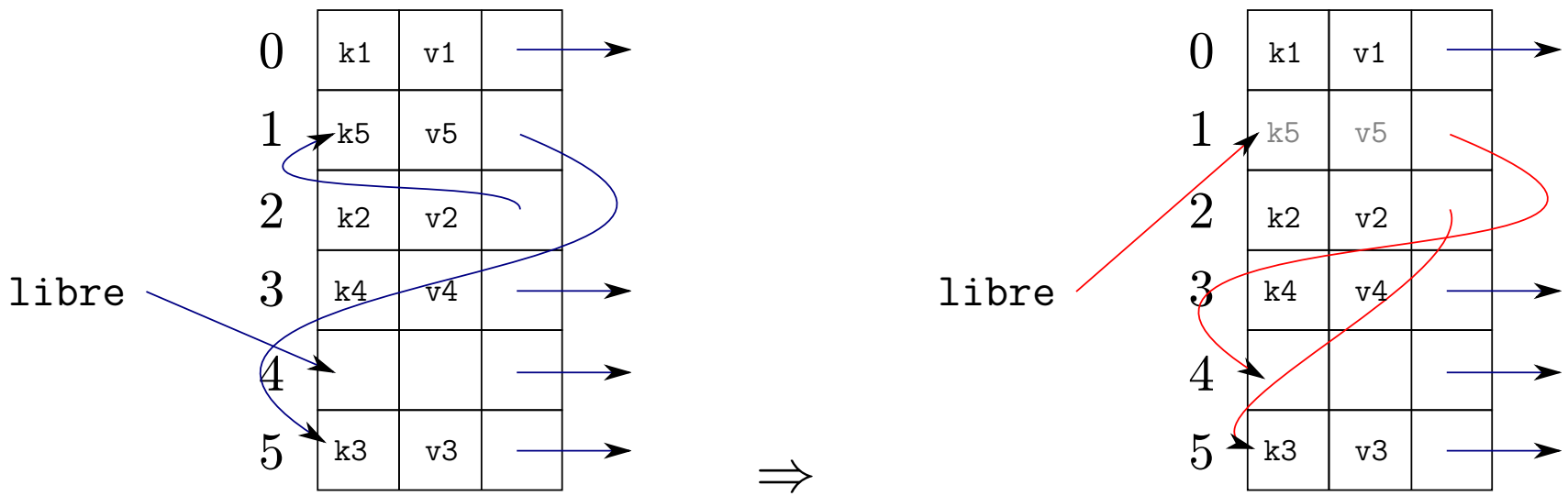
Après insertions/suppressions



Suppression : exemple

`supprimer(d,k5)`

- `hash(k5) = 2`



Suppression

`supprimer(d,k)`

- on trouve `k` en i
- on met à jour le prédécesseur éventuel en j :
  - on met comme successeur en j le successeur en i
- on libère la case correspondante :
  - on fait pointer `libre` vers i
  - on met comme successeur en i l'ancienne valeur de `libre`

Module paramétré

En OCaml, on a réutilisé les listes d'association pour faire les tables de hachage

- on ne peut utiliser que les fonctions déclarées dans l'interface de `Liste_assoc`
- on aurait en fait pu utiliser n'importe quel dictionnaire

Foncteur : module qui prend en paramètre une interface de module
$\simeq$ fonction sur les modules

```
module Table_de_hachage (D : Dictionnaire) =
  struct

    type ('k,'v) dict = (('k,'v) D.dict) array

    let rechercher d k =
      let h = hash k mod Array.length d in
      D.rechercher d.(h) k

    ...
  end

module TH = Table_de_hachage(Liste_assoc)
```